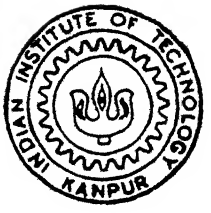
UNIX EMULATION SERVICES FOR MACH ON 386 AT

by

Prakash Bikkannavar



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March 1994

UNIX EMULATION SERVICES FOR MAC ON 386AT

A thesis submitted
in partial fulfilment of the requirements
for the degree of

Master of Technology

by

Prakash Bikkannavar

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

March, 1994

CSE-1994-M-BIK-UNI

CERTIFICATE

It is certified that the work contained in the thesis titled *UNIX EMULATION SERVICE FOR MACH ON 386AT*, by Prakash Bikkannavar has been carried out under my supervisio and that this work has not been submitted elsewhere for a degree.

March, 1994

Gautam Barua
Professor,
Department of Computer Science and Engineering,
IIT Kanpur

ACKNOWLEDGEMENT

I sincerely thank my thesis supervisor Dr Gautam Barua, firstly for giving me an opportunity to work in the field of distributed systems and secondly for his able guidance during the course of this work.

I am thankful to Krishna for the 386 ALP fundas and SaiRam for being so helpful all along. Special thanks to W. Jolitz for 386bsd source code. Stay at IITK would have been a tough experience had it not been for a handful of supportive and understanding friends to whom I was introduced during the last 18 months. I thank Vyasdya(of Vaikom), Anand, Kris, Shyam, Venkat,TJM for giving me some sweet memories to take from IITK, which I will cherish for a long time to come.

I also thank the CSE lab, library and office staff for all the transparent assistance they provided.

Most importantly, I am eternally indebted to my parents and my brother for their constant encouragement all through my educational career. To them I dedicate this work.

PrakashB

March, 1994

Abstract

This thesis describes the design and implementation of UNIX emulation services for MACH 3.0. The trend in operating systems research in the last few years has shifted from kernelization to dekernelization i.e, kernels enriched and overloaded with functionality and abstractions are steadily making way for microkernels. The basic goal is the development of a true distributed operating system.

MACH 3.0 is a microkernel developed at Carnegie- Mellon University. In IIT K, the development of PAWAN, a MACH based UNIX operating system began in 1992. MACH 3.0 was ported onto a M68020 based machine and a number of user level servers like file server, tty server, process server were developed. Later, the work on building emulation library began. This thesis is a continuation of the work. 386AT was chosen as the new hardware platform and the Mach 3.0 code for 386AT was used. All the servers developed before have been enhanced with certain design modifications and ported to 386AT. New services like pipes have been added. On top of these servers, the BSD 4.3 emulation an library has been developed. Currently, a near complete UNIX source code compatibility has been achieved. By developing sophisticated utilities, a full-fledged operating system can be built on top of it.

Contents

List of Figures

Chapter 1

Introduction

When UNIX was designed in the late sixties and early seventies, its design goals included: multiuser and multitasking capabilities, interactive use, time share operation, ease of use, structural simplicity and transparency, high functionality, especially suited for program development and documentation. An important decision was to put as little into the kernel itself as necessary and to implement rest of the system services like command interpreter as the user level programs. Implicit assumptions of the early design of UNIX were : single processor with rather limited memory, teletype like terminals, and networking consisting of serial interconnect of terminal ports. So the major advantages of the UNIX were its portability, simplicity and modifiability. With the enhancements in hardware technology, advent of distributed computing, multi-processors and growing needs for more sophisticated applications, lot of enhancements had to be made to the kernel. So over the years UNIX kernel has been modified to provide a staggering number of different mechanisms for managing objects and resources. Unix versions now support facilities such as system V streams, 4.2BSD sockets, network file system, pty's, various forms of semaphores, shared memory, and a mind-boggling array of ioctl operations on special files and devices. The result has been addition of scores of system calls and options with less than uniform access to different resources within a single UNIX system and within a network of UNIX machines. UNIX started losing its modifiability,extensibility and modularity. It became increasingly important to return to the original UNIX model of a consistent interface to system facilities. With this objective, the researchers started work to built newer operating systems using a collection of intercommunicating software modules or processes each performing a well-defined

function, such as file management, resource allocation or task scheduling. Development of easily configurable systems to which new services can be added without having to rebuild or restart existing system software, has been a positive step in this direction. Also, in such systems the boundary between the operating systems and the programs developed by users is less rigid, allowing the system to be viewed as an extensible set of software resources and services. Examples of such systems are the MACH, AMOEBA, V-system, CHORUS and the LOCUS distributed systems.

1.1 MACH Operating system

The MACH micro-kernel developed at Carnegie Mellon University is a successor to the ACCENT project and is intended as a basis for development of distributed systems that include multiprocessors. The Mach design anticipated many of the new computer hardware trends: it focusses on memory management, more precisely on managing large, sparsely populated virtual address spaces ; it stipulates hardware architecture independence and is designed for multiprocessor and distributed system support. MACH is fundamentally a message passing communication kernel. Operations on objects other than messages are performed by sending messages to ports. In this way MACH permits system services and resources to be managed by user-state tasks. The model provided by MACH is a client-server model in which objects are managed by servers and clients make requests for operation on objects by using remote procedure calls. MACH objects are represented by ports, which are protected message queues, and whose names can be transmitted in messages. By using Matchmaker, application programmers can be shielded from the intricacies of message composition, sending and reception and instead can be offered a procedural interface for sets of typed operations upon objects. The MACH virtual memory management system exhibits architecture independence, multiprocessor and distributed system support and advanced functionality.

1.2 Overview of Other Distributed Systems

1.2.1 The V Distributed system

The V distributed system [Cher84], is an operating system designed for a cluster of computer workstations connected by a high performance network. The system is structured as a relatively small "distributed kernel", a set of service modules, various run-time libraries and a set of commands. The kernel is distributed in that a separate copy of the kernel executes on each participating network node, yet separate copies cooperate to provide a single system abstraction of processes in address spaces communicating using a basic set of communication primitives. The existence of multiple machines and network interconnections is largely transparent at the process level. The service modules implement value-added services using the basic access to hardware resources provided by the kernel. The various run-time libraries implement conventional language or application-to-operating system interfaces.

The kernel is a software backplane that provides a base for building and configuring systems. It consists of lightweight processes and interprocess communication. New services are 'plugged in' and communicate with other processes using the IPC provided in the kernel and their internal design is invisible to the rest of the system. The V-kernel runs in each workstation and is based on clusters (V-teams) of threads, that can be dynamically created and destroyed and share an address space. Processes communicate via shared memory within a V-team and via network transparent, synchronous message passing between different V-teams.

1.2.2 AMOEBA

The AMOEBA architecture consists of four principal components [Rene89], [Mull85], [Coul88]. First are the workstations, one per user, which run window management software and on which users can carry out tasks requiring fast interactive response. Then we have the pool of processors, a group of CPUs that can be dynamically allocated as needed, used and then returned to the pool. The specialized servers such as directory server, file server and various servers provide specialised functions. Fourth are the wide area network gateways , which are used to link AMOEBA systems at different sites in possibly different countries into a single uniform system.

AMOEBA is an object oriented distributed system. The objects are abstract data

types such as files, directories and processes and are managed by server processes. A client process carries out operations on an object (transaction), by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object and sends a reply message back to the client which unblocks the client. To handle multiple transactions going on at the same time a process can be subdivided into lightweight subprocesses called threads. By having a thread for each request, a server process can handle multiple requests simultaneously. A client process can perform several transactions at the same time by having a thread per transaction.

1.2.3 The CHORUS Distributed System

The CHORUS distributed system enables us to integrate various types of operating systems - from small real-time systems to general-purpose operating systems, in a single operation system [Abro89],[Coul88]. CHORUS is a communication based technology. The kernel can be scaled to exploit a wide range of hardware configurations such as small embedded boards, multiprocessor workstations or high performance servers.

The CHORUS system is composed of three layers :

- A small real-time kernel (called Nucleus) integrating inter- application communications. The nucleus insulates the higher-level software from hardware and network specificites. The Nucleus provides local and global services:

  1. Local services : Fine grained synchronization and priority based scheduling of threads, local virtual memory management, exception handling.
  2. Global services : The IPC(Inter Process communication) manager provides the communication service, delivering messages regardless of their destination within a distributed system.

- A set of system servers complement the Nucleus with necessary high level services, in the context of Service assemblies. These services integrate the distributed nature of the architecture. Where necessary different assemblies may be built on top of the same nucleus. The main services which may be found in the assemblies are Process manager, File manager, Device manager etc.

- . Applications, mainly SWRU (Software replaceable units).

1.2.4 The X-kernel

The X-kernel is an experimental operating system for personal workstations that allow uniform access to resources throughout a nationwide internet: an interconnection of networks similar to the TCP/IP internet [Pete90]. Workstations are viewed as a portal through which users access both local and remote network resources. This is realized by providing an infrastructure that is general enough to support a wide variety of protocols, yet efficient enough that no protocol suffers a serious performance penalty. The X-kernel supports a library of protocols, and it accesses different resources with different protocol combinations. Two user-level systems have been built on top of the X-kernel to give users an integrated and uniform interface to resources. These systems, a file system and a command interpreter hide differences among the underlying protocol. The X-kernel incorporates components that manage processes, memory and communication. Multiple address spaces are supported, multiple light-weight processes can execute in each address space and processes within an address space synchronize using kernel supported semaphores. A communication manager is provided, which offers an object oriented infrastructure for composing protocols and a collection of powerful tools for implementing tasks common to all protocols.

1.2.5 The LOCUS Distributed System

LOCUS is an UNIX-like distributed system [Walk83],[Coul88]. It supports transparent access to data through a network wide file system, permits automatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, and is upward compatible with UNIX. The system provides a high degree of transparency concerning the location of files and some degree of transparency concerning the location of process execution. The system appears to clients like one giant UNIX system, with all the computers playing both client and server roles and with UNIX file access, process creation and interprocess communication primitives implemented transparently across the network. LOCUS is a procedure based operating system, processes request system- services by executing system calls, which trap to the kernel. The LOCUS file system presents a single tree structured naming hierarchy to applications and users. LOCUS names are fully transparent; it is not possible from the name of the resource to discern its location in the network. LOCUS provides nested transactions in which all

changes to a given file are atomic. To implement transactions, both original and changed data are kept at the storage site for the file in question, until a transaction is complete. LOCUS provides replication of files at the granularity of whole directories on the grounds that, if some node in a tree is inaccessible, then all the files below that node are inaccessible.

1.2.6 Development of MACH based UNIX system at IITK

Development of MACH 3.0 based UNIX operating system began at IITK in 1992 as part of M.Tech theses. The goal was to develop a full-fledged UNIX system that provides source level compatibility to applications. MACH 3.0 micro kernel was ported to a 68020 based mini computer. A number of user level servers like file server, process server, tty server were developed and some part of UNIX emulation library was developed. In this thesis work, the existing servers have been extended, debugged and ported to 386AT system. A user level Pipe server has been developed . The BSD 4.3 UNIX emulation library has been developed.

1.2.7 Outline of the Thesis

In Chapter 2, an overview of MACH 3.0 has been given. Chapter 3 discusses the various design alternatives along with the design of PAWAN. Chapter 4 deals with the tty server, chapter 5 deals with the file server, chapter 6 deals with the process server, chapter 7 deals with the pipe server. In chapter 8, the UNIX emulation services have been discussed. Chapter 9 concludes the thesis work.

Chapter 2

MACH Overview

This chapter presents a brief overview of the MACH kernel. The underlying philosophy, kernel abstractions and features of MACH are examined. More details on MACH can be found in the references [acce86],[Teva87],[Baro88].

2.1 MACH Philosophy

MACH has been designed with a goal of creating integrated computing environments, consisting of networks of uniprocessors and multiprocessors [Teva87]. The basic functionality of the kernel is designed to support the integrated computing environment of the future.

1. MACH supports diverse architectures (UMA, NUMA, NORMA).
2. It can handle range of communication speeds (LAN, WAN, tightly-coupled multiprocessors).
3. MACH is a new OS organization with
    - a small number of abstractions.
    - a kernel/OS server model.
    - network transparent and object oriented features.
    - integrated memory and communication.

The central component of a MACH-based operating system environment is the MACH kernel. Typical operating system services are layered above the MACH kernel as a set of

system servers. Since the MACH kernel is a simple base for system servers, it is readily portable and adaptable to the wide range of computing architectures present today and anticipated in the future. Since servers provide most of the traditional system services, different software environments can be run easily in different hardware environments.

2.2 Basic MACH kernel functionality

MACH can be viewed as being split into two components. The first is the small, extensible system kernel which provides scheduling, virtual memory and interprocess communications, and the second component is the several, possibly parallel, operating system support environments, which provide emulation for established operating system environments such as UNIX.

The MACH kernel supports five basic abstractions - Task, Thread, Port, Message and Memory Object.

- A task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory).

- A thread is the basic unit of execution. It consists of a processor state necessary for independent execution. A thread executes in the virtual memory and port-rights-context of a single task.

- A port is a simplex communication channel, implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in MACH. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports which represent them.

- A message is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

- A memory object is a secondary storage object that is mapped into a task's virtual memory. Memory objects are commonly files managed by a file pager, but as far as the MACH kernel is concerned, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data.

Message-passing is the primary means of communication both among tasks, and between tasks and the operating system kernel itself.

2.3 MACH Features

The MACH kernel functions can be divided into the following categories:

- task and thread creation and management facilities,
- virtual memory management functions,
- basic port and message primitives,
- operations on memory objects.

2.3.1 Tasks and threads

MACH divides the typical UNIX process abstraction into two orthogonal abstractions: the task and thread. MACH allows multiple threads to exist(execute) within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. The context switching time for threads of the same task is small. Operations on tasks and threads are invoked by sending a message to the task kernel port and thread kernel port. Threads may be created, destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within that task. In addition, tasks may be created and destroyed. A standard UNIX fork operation takes the form of a task with one thread creating a child task with a single thread of control and all memory shared copy-on-write.

2.3.2 MACH Virtual Memory (VM) Management

MACH has implemented a new, portable memory management system whose main features are:

- Architecture independence - support for a wide range of paged architectures [Teva87a].
- Distributed system and multiprocessor support - features such as shared memory and integrated memory management and message passing [Bolo87].

- Advanced functionality - especially, copy-on-write, shared libraries, memory-mapped files and user-implementable memory-objects.

MACH Virtual memory interface is divided into several functional groups:

- Address space manipulation including allocation and deallocation of virtual memory of a task at a page level.
- Memory protection allowing flexible use of different memory protection hardware.
- An inheritance mechanism for creation of address spaces in tasks.
- Miscellaneous primitives that formalize access to statistics maintained by the MACH kernel, access other task's virtual memory and describe a task's address space.

The new virtual memory design provides :

- Large, sparse address spaces (needed for large-scale AI application like speech, vision etc)
- Memory mapped files and user-provided storage objects (memory objects)
- Read/write and copy-on-write sharing (makes possible transparent parallel programming and networking).
- Integrated virtual memory and communication:
  - Large messages sent without physical copies being made (database management, graphics and AI).
  - Memory object capabilities can be passed in messages.
  - Network shared memory with variable consistency constraints.

An important feature of MACH's VM is the ability to handle page faults and pageout data requests outside the kernel. When VM is created, special paging tasks may be specified to handle paging requests. MACH also provides some basic paging services inside the kernel through a default pager task.

Memory allocated with no pager specified is automatically zero-filled and its page-out/pagein is handled by the default pager.

2.3.3 Interprocess Communication

The MACH interprocess-communication facility is defined in terms of ports and messages and provides both location independence, security and data type tagging. Ports are used by tasks to represent services or data structures. Access to a port is granted by receiving a message containing a port capability. Port capabilities include:

- send rights, which correspond to the capability to send a message to a port. Send rights may be held by any number of tasks.
- receive rights, which correspond to the capability to receive a message on a port. receive rights may be held by only one task.
- ownership rights, which correspond to the capability to gain receive rights on a port when the task having the current receive rights is terminated. It may be held by only one task.

The MACH kernel automatically queues messages for tasks executing on its machine. However, transmission of messages between separate MACH kernel hosts should be performed transparently by an intermediate server task, known as Network Message Server.

2.4 MACH Tools

2.4.1 MIG - the MACH interface generator

MIG is an interim implementation of a subset of the Matchmaker language that generates C and C++ remote procedure call interfaces for interprocess communication between MACH tasks [Drav89]. The MIG program automatically generates procedures in C to pack and send, or receive and unpack the IPC messages used to communicate between processes. The user must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates three files:

- User Interface Module: This module is meant to be linked into the client program. It implements and exports procedures and functions to send and receive the appropriate messages to and from the server.
- User Header Module: This module is meant to be included in the client code to define the types and routines needed at compilation time.

- Server Interface Module: This module is linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the generated interface module gathers the output parameters and correctly formats a reply message.

MIG is implemented as a cover program that recognizes a few switches and then calls cpp to process comments and preprocessor macros such as #include or #define. The output from cpp is then passed to the program migcom which generates the C files. The server procedures must be written by the user. The server waits for procedure call invocations on its communication port. To use the calls exported by the user interface module a client must first access the port to call the server on.

2.4.2 C Threads

The C Threads package allows parallel programming in C under the MACH operating system [Coop88], [Golu87]. The MACH kernel does not enforce a synchronisation model, it just provides basic primitives upon which different models of synchronisation may be built. The C-threads provides a high level C interface to the low level thread primitives along with a collection of other mechanisms useful in various parallel programming paradigms. It provides

- multiple threads of control for parallelism.
- shared variables.
- mutual exclusion for critical sections.
- condition variables for synchronization of threads.

The C-threads package in MACH has three possible implementations. The first implements threads as coroutines in a single task, the second uses a separate task for each C-thread, using inherited shared memory to partially simulate the environment in which multiple threads run. The third implements C-threads using MACH threads.

2.5 The I/O structure in MACH

MACH employs an I/O structure substantially different from that of existing systems. Peripheral devices are accessed through the device server port. Device server is implemented as a kernel object. The interface to the device server is through IPC. This section discusses the device server, its services and special features.

2.5.1 Device Switch

Device switch `dev_name_list` describes the devices potentially attached to the system. Each entry in this table describes the entry points to the driver. Another table Devs gives the bus specific description of devices. Noticeable differences from UNIX are:

- Integration of character and block devices in the same table.
- Kernel just houses the drivers and does initialization. It does not do any other operations on devices.

2.5.2 Normal Input and Output

Device server allocates a port for a device in response to `device_open()` request. This is the port which is then used to perform I/O on that device. Such ports can then be handed out to various tasks on the discretion of some trusted agent. It may be noted here that send rights to device server port imply complete control over all the devices, whereas access to the port corresponding to a device imply control over that particular device.

Normal I/O is done using MACH IPC on the device port. The I/O can be inband or out of band as suitable for specific devices, inband message passing being more suitable for small amounts of data. Out of band device I/O benefits from copy-on-write scheme used for message passing. It obviates memory to memory copying of huge amounts of data, thus improving efficiency.

Data is not buffered by the device server to avoid hard-wiring the policies within it. The applications which use its services are expected to employ their own buffering schemes.

2.5.3 Asynchronous Input

Asynchronous input is handled in a novel way in MACH through a special entry point in the device switch. This entry point `device_set_filter()`, associates a filter (a boolean function) with a port. The input from a device is then filtered and queued at the specified port if filter output is TRUE. This scheme makes it possible to have user level implementations of services which traditionally resided in a kernel e.g., network service.

This interface was used in an earlier design of the network server which had a user level implementation. The TTY server uses this interface for filtering special keyboard characters. For example, break key in cbreak mode is provided to the server in the control stream rather than the data stream - the filter recognizes it and acts in differently.

2.5.4 Iodone-processing

Functions performed at the completion of an I/O request are encapsulated in `iodone()`. In UNIX they generally consist of waking up the process waiting for it and are short enough to be executed from within the interrupt handler. MACH can not afford performing iodone-processing in the interrupt routine since it involves data transfer too. Hence `iodone_thread` is used to perform these functions.

Chapter 3

The Design of PAWAN

3.1 Introduction

Defining and standardizing a powerful micro kernel base is only the first step in realizing the potential of the overall MACH approach. The next step, and perhaps the one richest in design possibilities, is to learn how to construct a wide range of useful higher-level systems on top of this simple kernel. A particular class of such systems is the so-called emulation systems, that implement the application programming interface of an existing complete operating system or target system with various combinations of user level components viz. servers and/or libraries operating on top of the MACH kernel. The main benefits expected from this overall emulation approach include increased modularity, portability, flexibility, security and extensibility, as well as simpler development, debugging and maintenance[Juli92]. In this chapter, we discuss the different approaches to building operating systems on top of the Mach micro kernel. In the subsequent sections, the design goals of PAWAN and the design approach adopted will be explained.

3.2 Basic approaches

There are mainly two approaches to building operating systems on top of Mach micro kernel. Each of these approaches include the same three components: the `Mach microkernel`, one or more **system servers**, and an `emulation library`. The systems differ on how they provide the application system's personality - as a single integrated server supporting all operating system functions(the **single server** approach), or a collection of servers in which

each supports an individual function of the system(the **multi-server** approach).

3.2.1 Single server model

The *figure* 3.1 shows the single-server model, where UNIX functionality is implemented as one integrated server, providing all the UNIX abstractions from terminal handling to distributed file system support and network protocols. When an application program makes a UNIX system call, the micro kernel receives the call and immediately identifies it as for-the-emulator-not-me and bounces it back up to the UNIX emulation library. The emulation library receives the bounced call and transforms the requests into a remote procedure call to the UNIX server process. The UNIX server handles the incoming requests using a collection of threads.

3.2.2 Multi-server model

The *figure* 3.2 shows the multi-server model. In this approach, the UNIX system is partitioned into a collection of servers providing specialized functions and an emulation library translating UNIX abstractions into calls to the appropriate servers. UNIX becomes less of a complex monolith and more of a collection of interface specifications and abstractions translated by emulators and provided by servers. Developers could mix and match their UNIXes to specific needs. e.g., combining Berkeley fast file system with the system V interface etc.

Some servers may provide generic services and some other servers may be specific to a particular operating system. The high level services such as file management, network access, and local "pipe-type" interprocess communication, can be expected to be directly useful for a variety of operating system environments. In addition, many ancillary services that are not directly exported at the application program level, such as authentication, location of servers, configuration management etc., can also be reused in many different system configurations. On the other hand, process management remains largely tied to the semantics of each particular target operating system, mainly because of the complexity of the definitions of groups of emulated processes and associated rules of mediation. This approach definitely provides modularity, flexibility, and robustness of the system by isolating faulty or potentially malicious components into separate protected address spaces. Moreover it sometimes simplifies the implementation of some servers. However, such a desire for maximum modularity must be balanced against a number of practical considerations

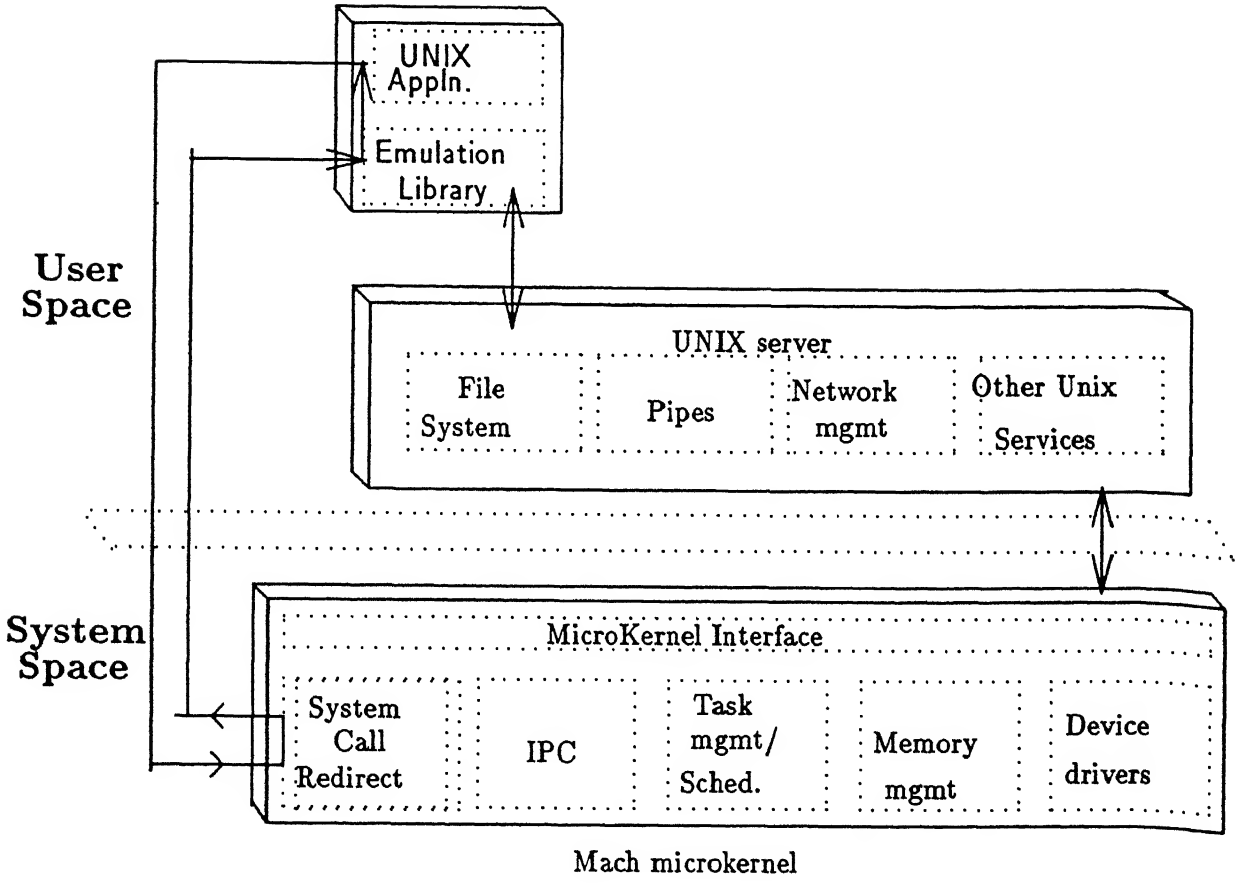


Figure 3.1: Single Server Model

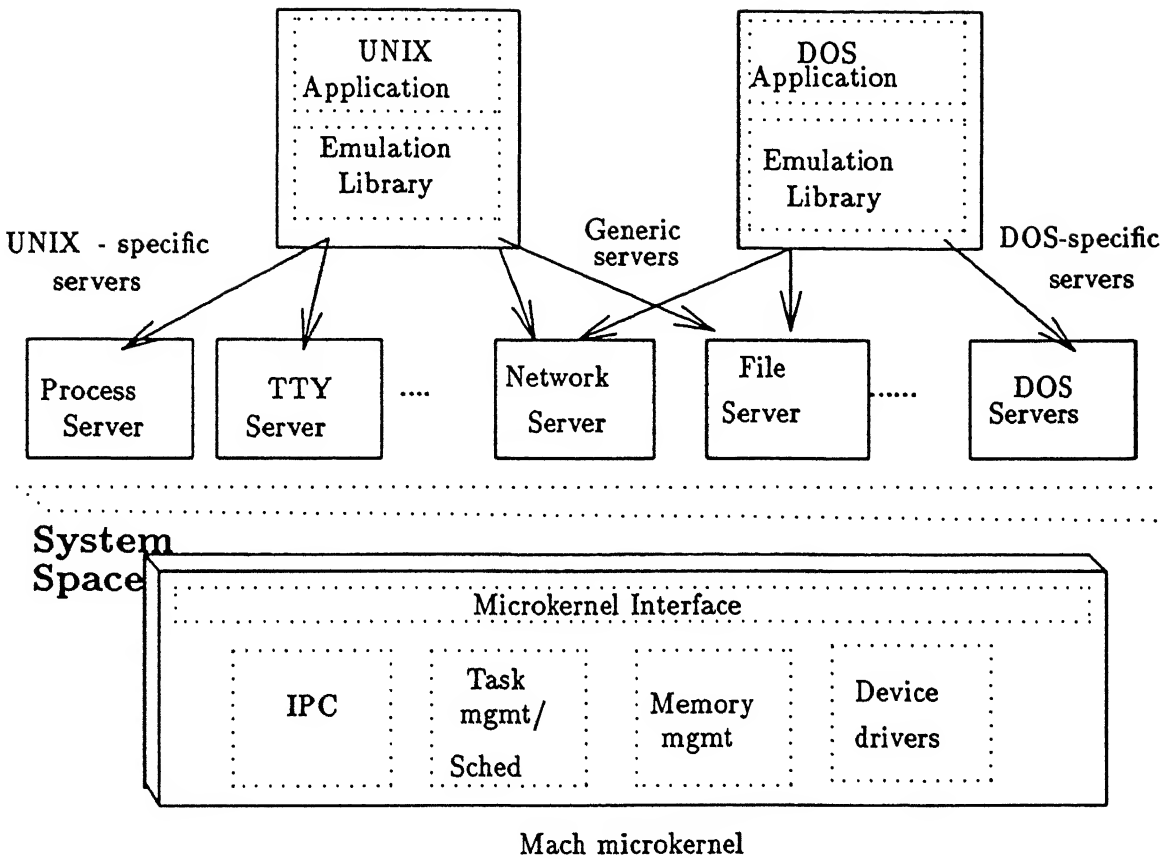


Figure 3.2: Multi-Server Model

[Juli92]. For example, interaction between servers are more expensive than interaction between modules inside a particular server. The separation of services and corresponding interfaces must be carefully defined to minimize those interactions.

3.3 Design goals of PAWAN

A major goal of PAWAN is to develop a set of servers and 4.3BSD UNIX emulation library using the services exported by these servers. Though the basic idea is to develop a set of generic servers that can be used for a variety of operating systems, our design of the servers has been been greatly influenced by the target operating system-BSD UNIX. The major driving factor behind the design of PAWAN is UNIX emulation and more particularly UNIX source code compatibility. Another goal has been to make use of the existing unix code wherever possible(e.g. File server). In all, Six user level servers have been developed. On top of these servers, 4.3BSD UNIX emulation library has been provided. These servers are the File server, the Process server, the Environment server, the Exec server, the pipe server and the tty server.

3.4 Description of servers

3.4.1 The File Server

The File server in PAWAN is a multi-threaded server and is fully compatible with the BSD file system. It provides UNIX-like file I/O and has been implemented by porting the 4.3 BSD file system code. It uses the MACH device server for device I/O. The file server has been designed to exploit the MACH features like multi-threading, out-of-line data copying, etc., to improve the performance. It maintains the file system related uarea information of all the tasks in the system.

3.4.2 The Environment Server

The Environment server facilitates the sharing of named variables between tasks. It is an extension of the *MACH EnvManager*. An environment is a set of named variables which can be read or changed via calls on an Environment port. Variables can be strings, ports or environments. An environment may be shared between parent or child tasks, or an

environment can be copied, and the copy can be passed to a child task. It is also possible to get a read-only port to an environment which allows reading but not modification of the environment. Variables stored in an environment are accessible through a specific server port. There may be one read/write and one read-only port to the same environment . Default or empty environments can be created with system wide constants useful to all tasks (like the hostname, network-address and well known server ports) and can be copied to one another. Ports and strings can be registered in or looked up from an environment by the tasks sharing it. A task can also use more than one environment: it could have access to a widely shared "global" environment as well as its own local environment. These features play an important role in system initialization and authentication. These issues are discussed later in this chapter. For detailed description of the Environment server please refer to [Gopa92].

3.4.3 The TTY Server

The TTY server regulates access to terminals and has been designed to serve any user interface in general. Although in UNIX, all device related services are put inside the file system, there are certain fundamental differences between user interface devices and a disk which make it advantageous to separate the code handling user-interface from the file system. It also provides facilities for job control, process groups and special character interpretation .The terminal driver inside the kernel was modified to use a special filtering mechanism provided in the MACH device server. This server has been explained in full detail in chapter 4.

3.4.4 The Pipe server

The pipe server has been designed to provide services for implementing UNIX pipes. The Mach IPC provides a reliable message delivery and also supplies flow control mechanisms for both readers and writers. Further, the implementation preserves the order of messages from single source. This, however, retains the message boundaries which is not the case in pipes. The pipe server makes use of IPC to implement the full semantics of UNIX pipes. Details of pipe server design have been given in chapter 7.

3.4.5 The Exec Server

PAWAN provides UNIX style *execve* which overlays the calling process by a new executable program. Pawan also provides *run* which creates a child process and runs the specified program in the child's address space.This is like *fork* followed by *exec* in UNIX. These are implemented as emulation library routines. However the implementation of setid exec is not possible at this level. We need to update the privileges, and this is possible only by the trusted servers. We run the Exec server with root privileges at startup time to take care of the setid exec. The execve and run library routines contacts the Exec server when the file to be run has its setid bit set. The Exec Server then takes up the responsibility of updating the privileges and running the program.

3.5 UNIX emulation library

The UNIX emulation library provides most of the 4.3BSD UNIX system calls. An attempt has been made to maintain the UNIX semantics strictly. In few cases the semantics have been redefined for various practical reasons. The library manages some information of the process, like the open file descriptors, signal state etc., at the fixed locations in processes's address space. The library routines basically translate the system calls into RPC's to the appropriate servers. The complete details of design and implementation have been given in chapter 8.

Chapter 4

The TTY Server

4.1 Introduction

Terminals are the user interface to a computer system. UNIX terminal support is through the file system which is inside the kernel [Bach86]. This leads to too many abstractions for the user to deal with, and therefore, too many system calls. Thus, the kernel becomes extremely cumbersome to upgrade. The advantages of the MACH philosophy of a small and fast kernel with all user-services offered through trusted, well-known servers has been stressed before. In PAWAN the tty server provides all the services related to terminals.

In UNIX, all device related services have been grouped inside the file system. Though this makes use of the file system code to give a common interface to all devices, there are certain fundamental differences between terminals and disks which make it advantageous to separate out the code handling terminals from the file system[Gopa92] The major differences are :

1. The TTY requests are always serviced sequentially, i.e., no access is stopped in the middle to allow another one to continue. Hence, the tty server need not worry about concurrent accesses to the same terminal, and can therefore afford to have just one thread per device waiting to serve requests. However, disk- scheduling and handling concurrent file system access are major issues to be handled .

2. A terminal cannot be used to store or retrieve information since any input/output it accepts is volatile. Therefore, crash recovery and consistency checks are not an issue in the tty server. However, these are of supreme importance in a file system.

Moreover, separating the tty server from the file server makes each server simpler and independently upgradable. A uniform UNIX like interface to access all device servers along with the file server can of course be provided in a separate UNIX emulation library.

The original design of PAWAN tty server was done by B.Gopal[Gopa92]. It has been extended and modified to meet the goals. The following sections describe the various design issues involved. Possible extensions to the tty Server are mentioned at the end.

The tty server has to support, apart from the basic read/write/ioctl operations, job control, process groups, special character processing, different modes like raw, cooked and cbreak.

The tty server forks off a thread for every open terminal. This thread listens on a particular port for user requests. Send right to this port is given to the user process opening the terminal. The tty server is stateful and needs to maintain the state of every user task accessing it, in a per-task Uarea identified by its tid (not to be confused with the pid which is the means to identify a task in servers providing UNIX services with complete UNIX compatibility). The uarea contains access control information (uids, gids, etc.), ports of the open terminals of the user, his pgrp number etc. One point to note is that on a fork, apart from asking the TTY Server to replicate its Uarea for its child, it should also give its child rights of all its open tty ports so that the child can access them with the same names.

Processes in UNIX are grouped into groups called pgrps. Every terminal being accessed has a pgrp number associated with it. All member processes of a process group with the same pgrp number as the terminal are termed as "foreground" processes, and the others as "background" processes. Only foreground processes are allowed to access the terminal, while special signals are dispatched to the background processes trying to do so. Pgrps have another function apart from access-control, namely, to process special character interrupts and provide UNIX-like signaling. These are described in the next subsection.

4.2 Implementation of read/write/ioctl

Mach kernel has a terminal driver which provides only the "raw" mode of input i.e., read returns if minimum of one character is in the input buffer even though read might have requested for more. There is no concept of special characters. This is too simplistic to implement the complete functionality of terminal interface and is clearly inadequate for our

purpose. Hence, various new features had to be added to the terminal driver.

The function *tty_ioctl* is used to set/get the terminal characteristics. The baud rate of serial communication, character length, software processing options of the input, software processing options of the output etc. form the characteristics of the terminal. There is a *termios* structure associated with each terminal that contain all these parameters.

```
struct termios {
    unsigned long       c_iflag;        /*  input flags   */
    unsigned long       c_oflag;        /*  output flags  */
    unsigned long       c_cflag;        /*  control flags */
    unsigned long       c_lflag;        /*  local flags   */
    char                c_cc[NCCS];     /*  special chars */
    long                c_ispeed;       /*  input speed   */
    long                c_ospeed;       /*  output speed  */
}
```

When a process sets terminal parameters it does so for all the processes using the terminal .The terminal settings are not automatically reset when the process that has set these exits.

When a process wants to write to a terminal, it makes a *tty_write* call to the read/write port that it got when the terminal was opened. The character buffer is sent to the tty server in-band as the data size is usually not too large. The tty server checks if the terminal is the control terminal for the process that has made the write request. If not, then the signal SIGTTOU is sent to all the processes in the process group. Otherwise it goes ahead with processing the output characters before writing to tty by making a call to the device server. The oflag (output flag) in the termios structure associated with the terminal decides how the characters should be processed. The most common processing done here is mapping NL to CR+NL, expanding the tab characters etc. After this processing the data is written to tty driver which displays the characters on the screen.

When a process wants to read from a terminal, it makes a *tty_read* call to to the read/write port of the terminal, giving the number of bytes to read. The tty server checks if the terminal is the control terminal for the process that has made the read request. If not, then the signal SIGTTIN is sent to all the processes in the process group. Otherwise

it sends a read request to the device server.

Typing certain special characters calls for special action to be taken. Unix supports three modes of input processing:

4.2.1 Line Mode (cooked mode)

In this mode, input characters are echoed by the system as they are typed and are collected until a carriage return is received. Only after a <CR> is received (or user request is satisfied before that) is the entire line made available to the shell or other process reading from the terminal line. In this mode special characters(like erase etc.) are processed. Some characters may generate signals sent to the current process associated with the terminal; these signals may abort processing or suspend it. Some other characters start and stop the output, flush output, or prevent special interpretation of the succeeding characters. The key issue is where to handle these special characters ? It could be either in the tty driver of the kernel or in the tty server. The Mach tty driver returns every character it receives. To handle cooked mode in the tty server it may require as many RPCs as the number of characters read. One RPC per character may be wastage of CPU time. An RPC consumes a considerable amount of time as it involves context switches apart from data copying. Moreover, on occurrence of some characters a signal has to be sent to the user process immediately; but, if the user is not doing a read operation,the special character typed will remain in the tty driver buffer and tty server will not know it till the user does a read operation. This is clearly unacceptable. There has to be a mechanism through which the tty driver can inform the tty server of the occurrence of a special character. MACH provides the following filtering mechanism for the kernel to inform the user of occurrence of asynchronous events.

1. Each device port (returned by device server on open) can have a set of filters associated with it.

2. A filter is an operation which returns whether an input matches (passes "through") it or not.

3. Each filter will have an array of characters and a reply port associated with it on which a message can be sent if the input matches with any character in the corresponding

filter. Device server provides the user an interface *device_set_filter/device_get_filter* to set/get the filter.

To handle the cooked mode, the tty driver has been modifiied. It returns from read, only if the user request is satisfied or a newline character is entered.The tty driver's function *ttyinput* is called by the COM driver interrupt service routine when a character is received. The *ttyinput* routine checks if the character is a special character. If so, it takes the appropriate action. e.g., if the special character is CINTR,CQUIT or CSUSP then it sends a simple message to the terminal's reply port on which a thread of the tty server will be waiting. On receipt of the character, this thread of the tty server dispatches appropriate an signal to all the processes in the process group of the terminal. If the character is not a special character then it is simply entered in the input buffer. The set of special characters and the reply port are set for a terminal using the call *device_set_filter* during the first open of that terminal and also during the subsequent *ioctl* calls.

4.2.2 One-character-at-a-time Mode

Unix supports two such modes :

1. **cbreak mode** : In this mode, the system makes each typed character available to be read as input as soon as the character is typed, and thus erase, werase, kill processing is not done. Other special characters are processed normally allowing signals and flow control as in cooked mode. This mode is set (through *ioctl*) by changing the filter entries corresponding to these characters to NULL.

2. **raw mode** : In this made, no character is processed. Every typed character is made available to the process reading from this terminal. This mode is set (through *ioctl*) by changing the filter entries corresponding to all the special characters to NULL.

4.3 Sequence of operations for open, read, write, ioctl

4.3.1 tty_open

1. Validate the user. If invalid user, then return error.

2. If the tty is being used for the first time:

- create tty structure. Call the device_open routine to open the com device.
- Create a device reply port. Set the default filter (default special character array along with the device reply port). Start a thread that listens on device reply port for some special characters.
- Create a user read/write port. Start a read/write thread which listens on this port for user requests.

3. Assign the process group appropriately.
4. Increment the open count of the terminal and return the send rights of the read/write port of the terminal to the user.

4.3.2 tty_read

1. Validate the user. If invalid user, then return error. Check if the process is a foreground process. If not send SIGTTIN signal.
2. Make a read request to the com device through the call *device_read_inband*.If it fails return error.
3. Process the read data as per iflag of the *termios* setting. The processing is usually converting NL to CR or vice-versa.
4. Return the read characters to the process.

4.3.3 tty_write

1. Validate the user. If invalid user, return error.
2. Check if the process is a foreground process. If it is a background process and if the termios setting disallows the background processes from writing then send SIGTTTOU signal to the process group of the process attempting to write.
3. Process the characters to be output as per the output flag of *termios* setting. This is usually mapping NL to CR-NL, expanding tabs to spaces etc.
4. Make a write request to the the com device through the call *device_write_inband*.If it fails return error.

5. Return the number of characters written.

4.3.4 tty_ioctl

1. Validate the user. If invalid user, return error.

2. Check if the process is a foreground process. If it is a background process then send SIGTTTOU signal to the process group of the process attempting to ioctl.

3. If the flavor is TIOCGETA then return termios settings of of the terminal. If the flavor is TIOCSETA then set the hardware related parameters and some parameters like ECHO/NOECHO by calling device_set_status. Set the filter for the terminal (array of special characters) by calling device_set_filter. Set the filter entries for the special characters CEOF, CEOL, CERASE, CWERASE and CKILL to NULL if the the mode is not canonical (i.e, if it is cbreak or raw).If the mode is raw(then set the filter entries for the characters CINTR,CQUIT and CSUSP to NULL. If the flavor is TIOFLUSH call the function device_set_status with the corresponding flavor.

4.4 Conclusions

The tty server along with the emulation library handles terminals with full UNIX semantics. Due to inherent serialization of all user interfaces, the TTY sever can be upgraded to handle any user-interface-device with a minimum effort if the corresponding driver is ported and integrated with the device-server.

Chapter 5

The File Server

This chapter describes the file system support provided in Pawan. MACH 3.0 is a microkernel and provides no file system support to the user. So a 4.3 BSD compatible file system has been designed and implemented as a user state server. UNIX file system's source code has been used extensively in the development of the file server. Our User File Server (henceforth referred to as UFS) runs along with other servers in the system and uses the kernel device server for device I/O. The server exports all the routines necessary to emulate the UNIX file system's system calls. The original design and development was done by P.V.Rao[Rao92] and D.Bairagi[Bair93]. This has been extended,modified at various places and has been ported to 386AT.

5.1 Design Goals

The main design goals of the file server are:

- Compatibility with BSD fast file system.
- To exploit the MACH features to improve performance.
- Multithreading for maximum parallelism.
- Security.

5.2 Design and Implementation

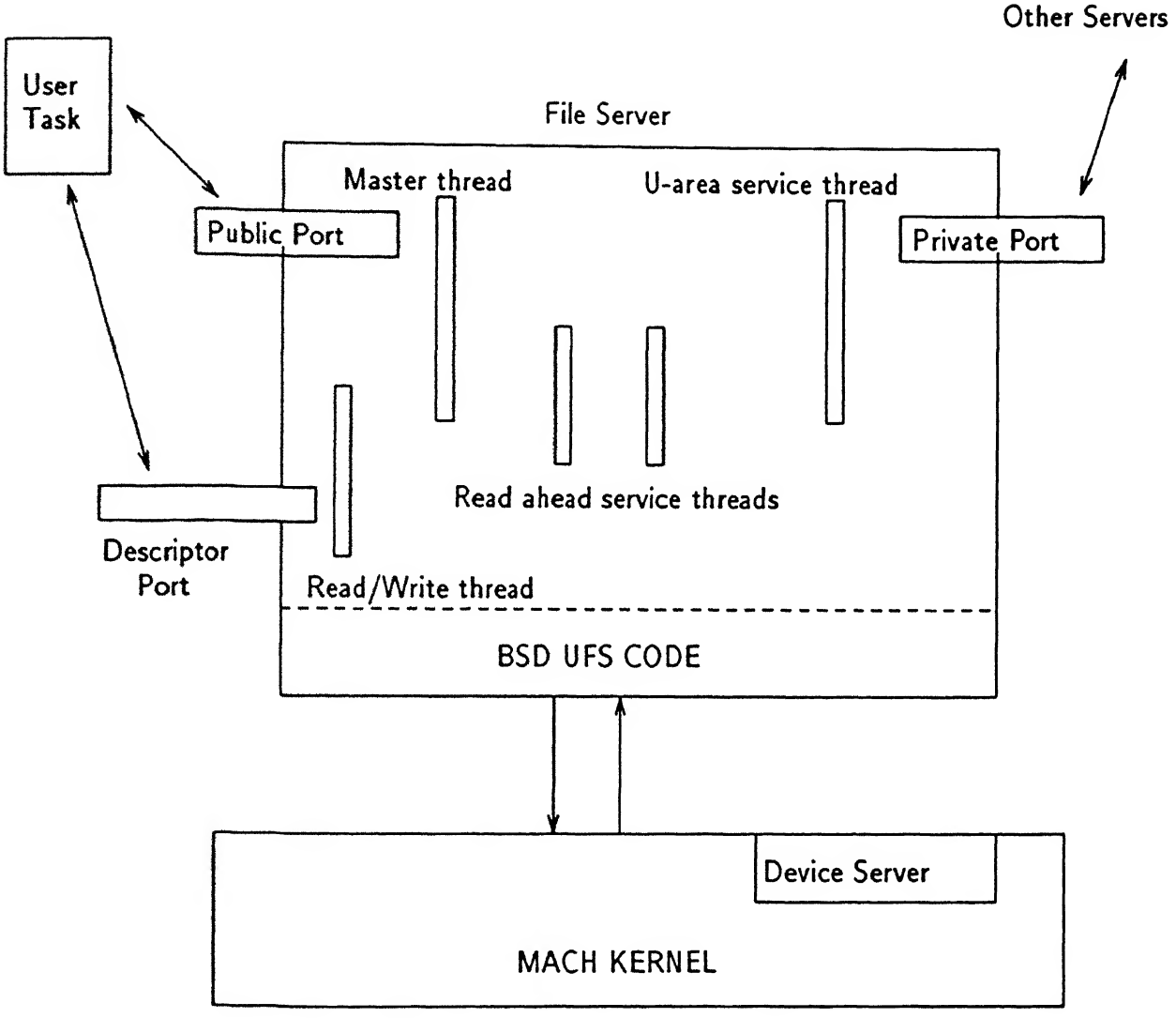


Figure 5.1: The PAWAN File Server

UFS has several threads of control. The master thread waits on its public port for users' file open requests. When a user task sends a file open request, the master thread returns a port as the file descriptor after opening the file. Read and write requests for the open file are sent to that descriptor port. A read-write(RW) thread, created by the master thread while opening the file, waits on the descriptor port for servicing these client requests. The uarea maintenance thread manages the per-task uarea of all the tasks in the system. It waits on its private port for requests from other servers for deletion,replication, and overwrite of uarea of any user task. A pool of read ahead service threads handle the requests for asynchronous read of disk blocks. The *fig* 5.1 shows the different modules of UFS.

5.2.1 The Uarea

As already explained, Uarea of any process is distributed in different servers. File server contains the uarea related to file operations. A user task in MACH typically contains multiple threads which can request file system services simultaneously. To service these requests in parallel, U area has to be split into a per request U area(U-request) and a per task U area(U-task). U-request contains mainly the parameters, return values and error code of the current request. U-task structure contains the fields common to all threads in a task. These include uid, gid, the file structures of the files opened by the task,current directory,home directory, namei cache etc.

5.2.2 Open Call

The master thread waits on the UFS public port for user requests to open the files. The user threads send requests for opening a file to this public port. An open request contains the file name, mode and the kernel port of the task that is requesting. The kernel port is used to identify the user task. On receipt of an open request, the master thread does the following sequence of operations:

1. Authenticate the user. Return error if unknown/unauthorized task.

2. Allocate the per-request uarea and retrieve the per task uarea. Set the parameters (file name and mode) for opening the file.

3. Open the file and store the *file* structure got into the free slot of the array of open files in the uarea. Store the index of this entry in a per-request uarea of the task.

4. Allocate a new port(descriptor port). Store this port in the array of descriptor ports in the per-task uarea. The index of this entry is the same as the index of the file structure. The descriptor port is associated (hashed) with the per-request uarea.

5. Fork a read/write thread which waits on this descriptor port to service all requests on this open file.

6. Return descriptor port's send rights to the user task.

5.2.3 Operations on Open Files

The user request for any file operation like read, write, lseek, fsync etc., on an open file is made on the descriptor port of that file. On receipt of the request, the read/write thread does the following sequence of operations:

1. Retrieve the per-request uarea structure which is hashed on the descriptor port. Retrieve the per-task uarea which is hashed on kernel port of the task that is making this request.

2. Call the appropriate routine to do the operation requested.

3. Return the result to the user.

If the operation is read, then the file server sends the read data to the User task out-of-line. Similarly, if the operation is write, then the user task sends the write data to the file server out-of-line. A file is closed if a user thread specifically requests for it or if the associated task dies without closing the file. If the reference count of the descriptor is more than one then it is decremented by one. If it is equal to one (i.e., if this is the last reference to the descriptor) then the actual close is done. The per-request uarea structure is deallocated. The descriptor port is deallocated and the read/write thread is killed.

5.2.4 Role of Uarea maintenance thread

The uarea maintenance thread waits on the UFS private port for the requests from other servers. The uarea maintenance thread accepts requests for creation of uarea of any task, duplication of uarea of one task to another and overwriting of uarea of any task. Request for creation of uarea for any task comes from the login server. The login server (yet to be

implemented) takes the login and password from the user, gets the uid,gid etc. from the password file, creates a new task, and then requests the uarea maintenance thread for the creation of uarea of this task.

In UNIX, Child processes created through fork inherit the open files of its parent process. Whenever a new process is being forked, the process server requests the file server's uarea maintenance thread to replicate the uarea. The uarea maintainance thread creates an uarea entry for the child process and replicates the parent's entry into it. Then it increments the referance coount of all the open files.

5.2.5 Read ahead mechanism

In UNIX, whenever the file system wants to read a disk block, it makes a call to the device driver and waits till the disk read is complete. If a file is being read sequentially, then the next read request is likely to be for the next block in the file. To improve performance, the read for the second block (called read-ahead block) is initiated soon after the first block is read. But it doesn't wait for this read to complete. Next time the request for this block comes, the block would already be in the buffer, and the read would return immediately. This asynchronous read of the read ahead block improves the performance significantly. After every synchronous block read, the file system initiates the asynchronous read of the read ahead block only if it feels that the file is being read sequentially. It makes the decision regarding this as follows. Every incore inode has a field that gives the block number in that file that was read last. If the current block number to be read in the file is x, and if the block read in previous read request of this file was x-1, then the block x+1 of the file is read asynchronously after block x is read. Otherwise only block x of the file is read.

In PAWAN the file server calls the device server to read a disk block. The device server calls are always synchronous, i.e. the thread calling the device server sleeps till the device server completes the call. So a single thread can not implement asynchronous read mechanism. A pool of read ahead threads(the number of threads is equal to the number of disks) are created when the file server is starting up. A queue is maintained for each thread. Whenever an async read of a block is to be initiated (in breada), the (locked) buffer address of the block is put in the appropriate queue and the corresponding read ahead thread is informed. The main read/write thread continues. The read ahead thread gets the request from its queue and makes a device_read call to the device server. If the queue is empty, then

it sleeps till the queue becomes non empty. The synchronization is done using condition variables and mutex locks.

5.2.6 Synchronization and handling of critical code

In UNIX, a process is never preempted inside a system call to maintain consistency of the kernel data structure. In PAWAN, since UFS runs as a user task, a RW thread may be preempted and another RW thread scheduled while the first thread is manipulating some global data structure(critical code). This may lead to data inconsistency. To avoid this, the threads in UFS co-operate within themselves and maintain the consistency of the global data structures. A mutex variable exec_mutex is used for this purpose. Any thread wanting to execute the critical code has to lock this exec_mutex first. Those threads which want to execute this code but cannot acquire the exec_mutex lock, will go to sleep.The Condition variables buf_free, ino_free are used for sharing of buffers and inodes in UFS. Buffers are locked by setting a flag in the buffer by the thread that is using them. Any thread trying to use a buffer checks whether it is locked or not. If it is not locked the thread locks it and goes ahead. If it is locked, the thread does a condition_wait on buf_free condition variable. Threads do a condition_signal on buf_free when unlocking a buffer, waking up any thread waiting for buffers. Similarly access to the inodes is shared.

Block Size	Read		Write	
	UNIX	PAWAN	UNIX	PAWAN
8K	27 msec.	53 msec.	20 msec.	54 msec.
4K	12 msec.	35.6 msec.	3 msec.	16 msec.
2K	6 msec.	19 msec.	1.25 msec.	11 msec.

Figure 5.2: Read/Write timings of PAWAN file server and 386BSD file system

5.3 Performance Evaluation

Performance of the file server was compared with that of the 386bsd file system. The granularity of time measurement in MACH is 1 tick which is 10 msec. So, to measure the read/write timings, the read/write operation was repeated a number of times in a loop. The operation was essentially a sequential read/write, and was done on a huge file. The absolute times were recorded(using *host_get_time* before entering the loop and soon after coming out of the loop and average read and write timings were calculated. A similar thing was done for 386BSD, where the granularity of time measurement is 16.6 msec. This operation was repeated with different sizes of data. Figure 5.2 shows the average timings for read/writes. It shows that the performance is quite poor as compared to UNIX.

Every read operation involves one or two RPCs. One RPC is from the user task to the file server. There will be another RPC from the file server to the device server if the data is not available in the file server buffers. The user gets the data from the file server out-of-line. So, the user(library) has to copy this data to the buffer sent by the read call and deallocate it. It is this copying that is expensive. e.g, copying a block of 8K from one buffer to another (using *memcpy*) takes about 9 msec. There is any way another copy from a file system buffer to the message buffer. Thus, there are two copies, versus one in UNIX. The performance of write is also poor. This may be due to a number of reasons. First of all, there are two copy operations as in the case of read. Then there is the overhead of RPC. The effectiveness of write-behind depends on the number of buffers in the file system which is smaller than in 386BSD in the current configuration of MACH.

Clearly, there is a need to tune the system. The extra copies have to be avoided. One way could be to avoid the copy in the file server for read operation. The file server buffer containing the data can be sent directly out-of-band to the user task, which will then copy the required data into the user variables (this will then be the only copy operation). During write, the pages of the user task containing the data to be written is to be mapped to the file server address space, where a copy to the file system buffers can take place (the only copy operation).

5.4 Conclusions

The file server exports all the necessary routines needed for the complete emulation of the BSD file system. As explained in the previous section, the file server performance is quite poor compared to 386bsd file system. There is need to idenify the bottlenecks and take corrective measures. File locking, and select calls are not implemented yet.

Chapter 6

The Process Server

6.1 Introduction

A process is a program in execution. A process must have system resources, such as memory and the underlying CPU. Unix supports multitasking - illusion of concurrent execution of multiple processes. User can create processes, control the processes' execution, terminate the processes. Each process is identified by a unique number called Process Identifier(*pid*). A process may create a new process that is a replica of original by using the *fork* system call. The new process is termed as child process of the original parent process. The context duplicated in process creation includes both its user-level execution state and system level state. The fork call reurns twice, once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is zero. The parent-child relationship induces a hierarchical structure on the set of processes in the system. The new process shares all its parent's resources, such as file descriptors, signal handling status, and memory layout. A process can overlay itself with a new executable file using the system call *execve*. A process may terminate by executing *exit* system call, returning eight bits of exit status to its parent. A process can suspend execution until any of its child processes terminate using the *wait* system call, which returns its pid and exit status. A parent process can arrange to be notified by a signal when a child process exits or terminates abnormally. Signals is a facility used inform processes of occurrence of asynchronous events. In PAWAN, the process server provides the necessary support for the implementation of all these process related calls including the signals. In the next section, we describe the design and implementation of the process server.

6.2 The Design Goals

The process server was designed to provide the following functionality:

- Support for creation, controlling the execution and termination of processes.
- Provide unique process identification and maintenance of process related information(stored in the process table and uarea in Unix).
- Support for Unix signalliing and process synchronization.
- Security issues.

At the relevant places, we have described why we tried to provide the above functionality in the process server. The process server design is an extension with certain modifications of the Signal server design proposed by Ashish Singhai[Ashi92].

6.3 Process Server Design

In MACH a task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and port rights. A thread is the basic unit of execution. It consists of all processor state (e.g. hardware registers) necessary for independent execution. A thread executes in the virtual memory and port rights context of a single task.

Traditionally, a Unix process has a single thread of control. This notion of a process is, in MACH, represented by a task with a single thread of control. So, in PAWAN, we decided to have a thread(called Unix thread) in a task to represent a UNIX process. (Actually every Unix task will have another thread called signal thread that is used to implement signals, as explained later in this chapter)

A Unix process is much more than a task and a thread of control. It has information about the open descriptors, information about how the process wishes to react to signals, owner's identity, event descriptor, accounting information etc. Unix stores this information in the kernel in uarea and process structure. In PAWAN, this information is stored partly in the emulation library. The remaining part of the process information is distributed among a number of servers apart from the process server. File server, for example, stores information about the open files of the process. The process server stores the pid, event

descriptor, signal related information etc. Thus the "uarea" of a process is distributed among a number of servers and is coordinated by the process server. The entry for a task is made by the Process server when it receives information about the creation of a new task (via the *task_create_msg* server call). The server assigns a unique pid to the task, associates the correct parent pid and duplicates rest of the information from the parent task's entry. Then process server requests the other servers (on their private ports) to make an entry for the new task and duplicate the relevant information from it's parent's entry. For example, it requests the file server to duplicate the open file descriptor table. Whenever a process is being terminated(by calling *exit* routine), a process server call *task_terminate_msg* is made. The process server cleans up the information about the process, it requests other servers to cleanup their part of the process information. Then it does other things like sending the SIGCHLD signal to the parent, storing the exit status etc. Finally, it destroys the task. Note that the entry for the process is not destroyed till its parent gets the exit status using *wait* system call.

6.4 UNIX Signals

Unix Signals inform processes of occurrence of asynchronous events. The system defines a set of signals that may be delivered to a process. Signals in 4.3BSD are designed to be software equivalents of hardware interrupts or traps. A process may specify a user level subroutine to be a handler to which a signal is to be delivered. A process may, instead, specify that a signal is to be ignored or that a default action as determined by kernel is to be taken. A process may receive a signal from either another process or kernel. The kernel sends signal to a process in case the process generates an exception due to execution of illegal instruction or tries to access illegal memory locations. UNIX Signals can be classified under following groups.

- Signals having to do with termination of a process.
- Signals having to with process induced exception e.g. execution of illegal instruction.
- Signals caused by an unexpected error condition during a system call.
- Signals originating from a process in user mode.
- Signals related terminal interaction.

- Signals for tracing execution of a process.

Some signals cannot be caught or ignored. These include SIGKILL, which is used to kill runaway processes, and the job control signal SIGSTOP. All signals in UNIX have the same priority. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific.

Basic interface to signal facilities in BSD4.3 is through kill() and sigvec(). Kill() is used to send a signal to specified process and sigvec() to specify the action to be taken on arrival of a signal. Action could be one of default, ignore or catch. If the action is catch then a function which is to be called when the signal is delivered is specified. The default action as decided by kernel can be one of ignoring the signal, terminating the process, terminating the process after generating core file or Stopping the process.

UNIX signaling is split into two parts - posting a signal and delivering it. Signals are processed in the context of receiving process. Posting a signal principally consists of adding the signal to a process's set of pending signals. Process is also set to run unless sleeping at non interruptible level. Signals are detected when a process returns from sleep, in ast handler, or when a process returns from a system call. In each of the above cases a call to the function *issig* is made which checks if a signal is pending, if so it arranges to invoke the handler or take default action. If user level handler is to be called then the process state so manipulated that, on return to user mode, a call is made to the handler immediately .

6.5 Signals in PAWAN

There are some factors that make signalling mechanism unnecessary in a MACH based system. MACH provides an IPC interface which is very convenient for most of the functions that signals perform in UNIX. Furthermore, as MACH is a multi-threaded system there is no need to complicate the asynchronous event notification by stopping the user thread and invoking the handler; a separate thread can wait for the event to occur. Similarly blocking/ non-blocking send and receive operations render signals unnecessary for synchronization.

In PAWAN, we have provided a UNIX style signalling facility. Signal handling is done partly in the emulation library and partly in the process server. An important factor to note is that a even user level implementation should not allow runaway tasks to defy SIGKILL. This necessitates some form of kernel support viz. having access to a task's kernel port.

The need for a reliable implementation as well as the need for some sort of kernel support (to implement SIGKILL and handle the default processing) induced us to go for a server based implementation of the UNIX signalling facility.

The key idea is to have a signal thread in every unix task and implement the signals by message passing. Here we take advantage of the MACH kernel property which allows a task to make kernel calls on behalf of another task if it has access to the task's **task kernel port**. Thus we can make kernel calls requesting it to terminate, suspend or resume other tasks. The kernel port of all emulated processes are sent to the process server via the *task_create_msg* during the process creation. At the same time, a new thread called signal thread is also created. This thread starts listening on the task's *exception port* for signal messages.

A signal to a process may originate from either another user process or from kernel. It originates from another process when that process makes a *kill* system call. It originates from kernel when the Unix thread generates an exception due to execution of illegal instruction or illegal memory access etc.

Following is the sequence of events when a process A sends a signal to process B:

- Process A makes a kill_task(A, B, signo) call to the process server's public port.
- In Process server: The process server receives the signal request. It does privilege checks. If A does not have necessary privilege then an error is returned. If the signal can not be caught then default action is taken. If the signal is SIGCONT and the process is stopped then it is resumed. Its status is set to RUNNING and a signal SIGCHILD is sent to the parent. Otherwise, signal has to be delivered to the task B. It sends a message to the signal port of the process B using the call *unix_signal_raise*. If the signal is being handled/ignored then the signal thread takes care of it and the call returns success. If the action to be taken for the signal is default action, then the call returns return with a flag. In this case the process server takes the default action on the process B. The default action can be terminating the process B (either with core or without core), stopping the process or ignoring the signal. The process server can stop/terminate any task as it has access to the kernel port all the tasks. If the default action is stop then it is stopped. If the default action is terminate then the process is terminated. In both these cases the process status is set so that the

parent doing wait() can get the status. The signal SIGCHLD is also sent to the parent process in both these cases.

- In the signal thread of process B: The Signal thread, waiting on the exception port of the task receives the signal number. If the signal needs a default action (i.e it is not handled /not ignored) then it returns failure to the process server which will take the default action. Otherwise it adds the signal to the list of pending signals. Before going to sleep on *receive*, it deals with all the pending signals. It takes one pending signal at a time. If the signal is being ignored then it does nothing. Otherwise, it suspends the unix thread and calls the handler. Once the handler returns then the UNIX thread is resumed.

A faulty user program may try to execute an illegal instruction, or try to access a memory location that it access to, then the machine generates a hardware exception. The MACH kernel catches this exception. If there is no thread listening on the exception port of the task that has caused the fault then it calls kernel debugger. Otherwise it sends a simple message to the exception port of the task, giving full information about the fault (task, thread, exception code, subcode). The signal thread that is waiting on the exception port receives this information. It finds out the signal that corresponds to this fault. If the default action action is to be taken then it informs the process server to do so as default action can be taken only by the process server. If the signal is being ignored then it does nothing. Otherwise, it suspends the unix thread and calls the user handler. Once the handler returns then the UNIX thread is resumed.

6.6 Authentication Server

Security is one of the most important issues in the design of an operating system. The basic goal is to authenticate the system user and prevent unauthorized usage of files, communication channels and other objects. In UNIX, the kernel associates two user id's with a process, the `real uid` and the `effective uid` [Bach86]. The real uid identifies the user who is responsible for the running process. The effective uid is used to assign ownership of newly created files, to check file access permissions, and to check permissions to send signals to processes via the *kill* system call. The kernel allows a process to change its effective uid when it executes a setuid program or when it invokes the setuid call explicitly.

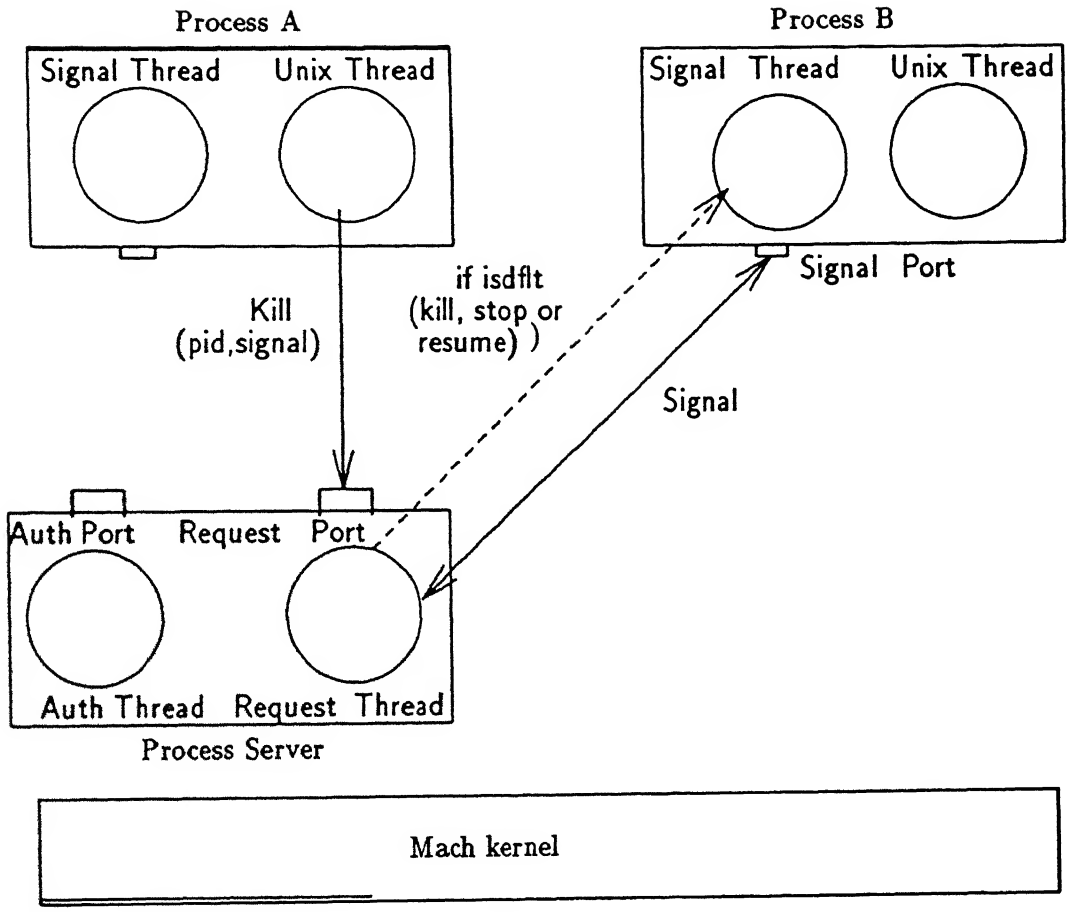


Figure 6.1: Signal sent from one process to another

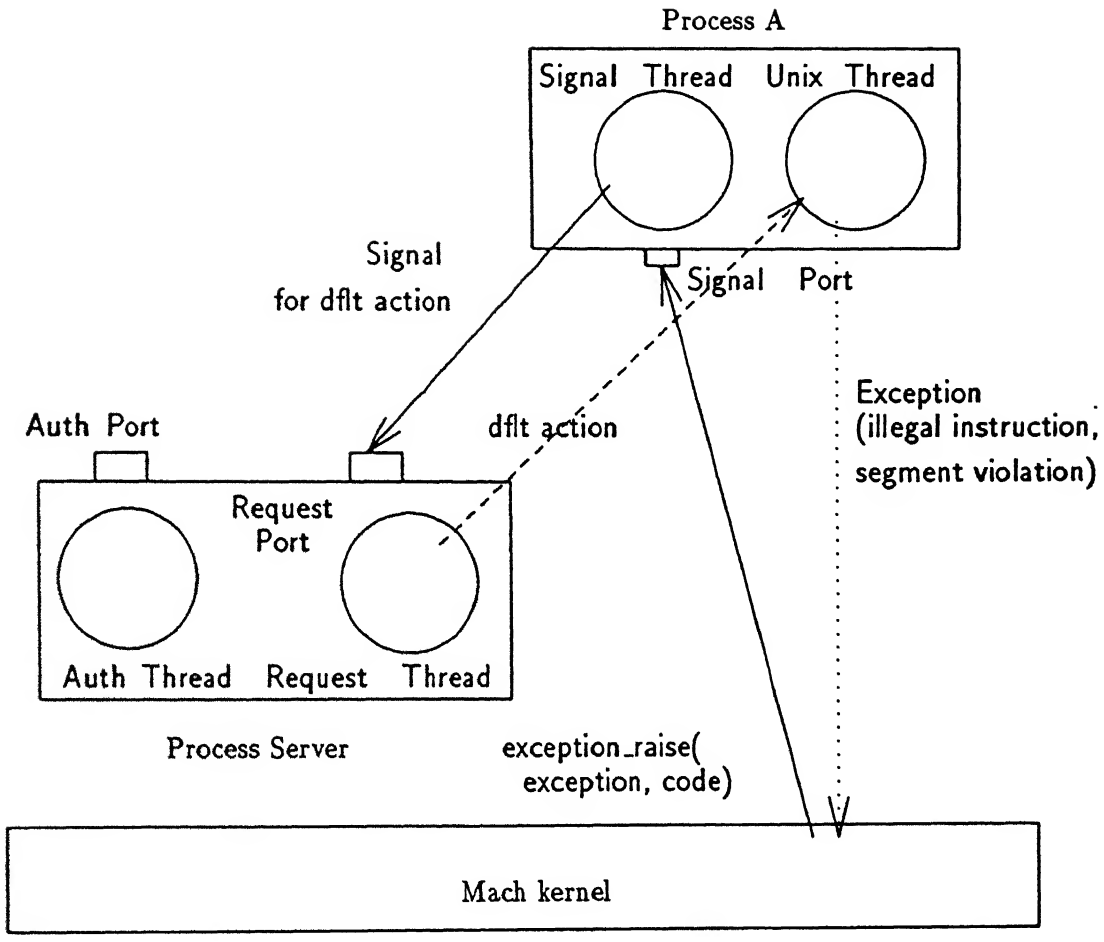


Figure 6.2: Exception generated by a process

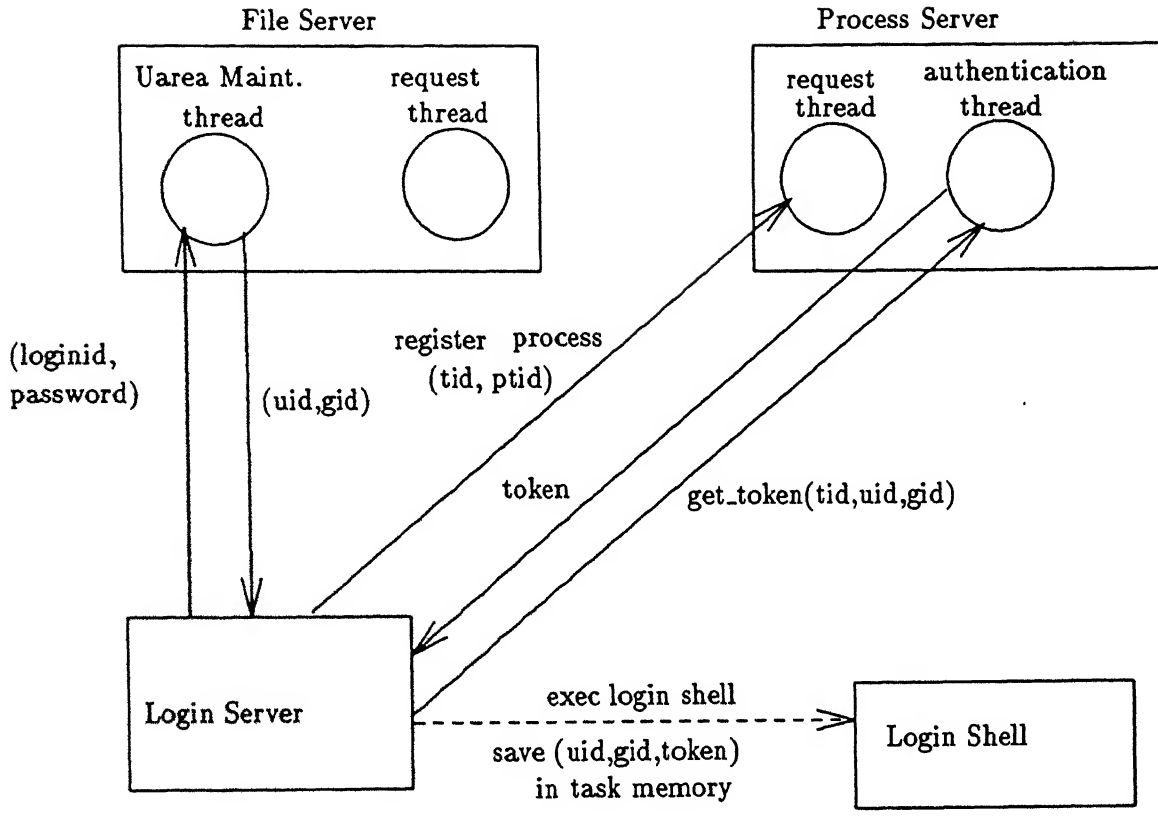


Figure 6.3: Initialization Of Credentials

In PAWAN the state of the system is effectively distributed among the various servers and the emulation library . The real and effective uid's are stored in each of the trusted servers, which use it to authenticate the requests. This information is also kept by the emulation library. The privileged calls (involving changing the server state, like changing access rights of a user) are allowed only on the private ports of the servers. Since these ports are accessible only to other trusted servers, the data within the servers is protected against violation by the user. The level of protection provided with just this mechanism is not enough to provide UNIX style authentication. The decision to place more responsibility for various system functions in an emulation library that is not protected from incorrect or malicious user programs has implications in the area of robustness and security. A malicious client must be prevented from gaining access to protected resources. Authentication server runs as a thread of the process server. This thread will be waiting for the requests from other servers on its private port. The kernel port of a user task is used to identify a task. This port can be destroyed or even swapped, but never actually forged. An unique token is assigned by the authentication server when a new user logs in. The token along with the user id and group id is stored in fixed memory locations of the user process. Any user while requesting for a service from a server, shows its token to the server. The server gets the true token of the user by contacting the authentication server. It caches in this token for future requests. If the token as given by the user matches with the one got from the authentication server then the request is entertained otherwise it is rejected. Figure 6.3 shows the initialization of credentials during the logging in of a user. The uid, gid and token of a process are passed on to all its child processs that are subsequently created. A new token is assigned by the authentication server when a new set of credentials is to be installed for a task. This is usually done during the execution of the setid programs.

Chapter 7

The Pipe Server

7.1 Introduction

Pipes is an interprocess-communication facility that supports unidirectional flow of data between related processes. Data transfer is stream oriented, reliable and flow controlled. They also allow synchronization of process execution. Their implementation allows processes to communicate even though they do not know what processes are on the other end of the pipe. There are two kinds of pipes : *named pipes* and, for lack of a better term, *unnamed pipes*. They are identical except for the way that a process initially accesses them. Processes use the open system call for named pipes, but pipe system call to create an unnamed pipe. After that, the processes use the regular system calls read, write, and close when manipulating pipes. Only related processes, descendents of a process that issued a pipe call, can share access to unnamed pipes. However, all processes can access a named pipe regardless of their relationship, subject to the usual file permissions. In this chapter, we discuss the design and implementation of unnamed pipes in PAWAN.

7.2 Implementation of Pipes

In System V UNIX, Pipes have been implemented as part of the file system. Since a pipe doesn't exist before its use, the kernel assigns an inode for its creation. It uses the file table so that the interface for read, write, close and other system calls is consistent with the interface to the regular files. The byte offset is stored in the inode unlike the regular files where it is stored in file table. This allows convenient FIFO access to pipe data. The

maximum size of the pipe is the amount of data that can be stored in inode direct blocks.

In BSD UNIX(version 4.2 onwards) the pipes have been implemented using stream sockets[Leff89]. Pipes are really a special case of the socketpair system call and, infact, are implemented as such in the system.

In PAWAN, a dedicated server has been implemented to handle pipes. The emulation library gives the actual system call interface to the user. The MACH kernel IPC facility provides message based, capability based IPC which has the following properties :

1. In-order delivery of data.
2. Unduplicated delivery of data.
3. Reliable delivery of data.
4. Flow control
5. Preservation of message boundaries.

Pipes have all these properties except the fifth one. i.e., there are no message boundaries in pipes. So MACH IPC can be conveniently used to implement the complete semantics of pipes.

The pipe server manages information about all the open pipes in the system and and also the per task open pipe information. Pipe data is stored as a circular buffer. The maximum pipe size is 4096 bytes, but is configurable. Mutex lock and condition variables are used for synchronization. There are two main threads in the pipe server. One is *request thread*, and the other is *uarea thread*. The request thread waits on the public port for the users requests to open a pipe. The uarea thread waits on the private port to handle requests from other servers for manipulating the per process uarea.

On receipt of *pipe open* request from a user, the request thread does the following sequence of operations:

1. Create a new pipe structure.
2. Create two new ports: read port and write port. Associate(hash) the pipe structure with both the read port and write port.
3. Add to the task's list of open pipes.

4. Create two threads: read thread and write thread. Read thread waits on the read port for the read requests on that pipe. Write thread waits on the write port for the write requests on that pipe.

5. Return send rights of read port and write port to the user.

To read from a pipe, user sends the *pipe read* request to the read port of that pipe. On receipt of read request, the read thread extracts the pipe structure corresponding to the read port. If the buffer has data then it returns as many as requested. If the buffer has less data than requested, then it returns as much data as it has and awakens any process that is waiting for the pipe to become non-full. If the buffer is empty then it goes to sleep on a condition variable. When awakened by writer thread it again tries to read the data from the buffer. It return zero bytes if there is no writer process and the pipe is empty.

To write to a pipe, user sends a *pipe write* request to the write port of that pipe. The data is sent to the pipe server *in line* or *out of band* depending on size of the data. On receipt of write request, the write thread extracts the pipe structure corresponding to the write port. If the pipe is full, then it goes to sleep till the pipe becomes non-empty. If there are no readers, then the SIGPIPE signal is sent to the process trying to write, otherwise, it writes data to the pipe buffer.

To close a pipe descriptor port(read or write), user sends a close request to that port. On receipt of the close request, the corresponding thread extracts the pipe structure for that port. The reference count of that port is decremented. Now, if both the read and write port reference counts are zero for that pipe, then the pipe structure is deallocated, read and write threads are killed, and the ports are deallocated. If only the read port reference count has become zero, then all the sleeping writers are awakened. If only the write port reference count has become zero, then all the sleeping readers are awakened. The user's uarea is updated to indicate closing of this port.

When a process forks, the child inherits all the open descriptors. So, during the *fork* call, the process server requests the pipe server to make an entry for the new process and copy open pipe descriptors from the parent. The uarea maintenance thread, which will be waiting on the private port receives this request. It replicates the open pipe descriptors from parent to the child and increments the reference counts of all these ports.

7.3 Conclusion

Full UNIX semantics of the pipes has been implemented successfully. The emulation library provides the common read/write interface to pipes and other objects referenced through descriptors.

Chapter 8

PAWAN Emulation Library

8.1 Introduction

In the previous chapters a set of servers developed on top of the MACH kernel have been explained. These servers include file server, process server, tty server and pipe server. Though some of these servers were designed to provide generic services, the design was greatly influenced by the target operating system BSD UNIX. On top of these servers, emulation library can be built for any operating system. The emulation libraries take the application requests and route them to the appropriate servers. The emulation libraries communicate directly with each server without requiring centralized co-ordination. As a result, the state of the system is distributed among the various servers. In the subsequent sections, we describe the implementation of the 4.3BSD UNIX emulation library.

8.2 File System Interface

UNIX provides a common interface to regular files, devices, pipes and sockets. In fact devices (like terminals) are considered as special files. Unix processes use descriptors to reference these. A descriptor is a small unsigned integer. A read or write system call can be used on the descriptor to transfer data. The close system call can be used to deallocate any descriptor. Kernel keeps a descriptor table for for each process. Descriptors represent the underlying objects supported by the kernel and are created by system calls specific to the type of object. The system call *open* is used for opening the regular and device special files. The system call *pipe* is used to create a pair of pipe descriptors. The system call *socket* is

used to open a socket etc. Then there are some system calls specific to a descriptor. For example, lseek, fstat etc., are meaningful only to file descriptors.

Descriptor management

In PAWAN, every descriptor is a pair of two unsigned numbers (descriptor_port, descriptor_type). The descriptor_port is the name of send rights to the port got from open, pipe or socket call and the descriptor_type denotes the type of the descriptor. The type could be pipe, file, tty, socket etc. The descriptor array of every process is stored in fixed locations in the address space of the process. This makes it convenient to access the descriptors and also to pass on the descriptors to child processes on execution of fork or run.

Implementation of system calls

The system call *open* sends a request to the public port of the appropriate server for opening the particular file/device. If the server manages to open it successfully, it returns send rights to a port on which it will be a waiting for the user read/write requests. This port along with its type is stored in the descriptor table. The index of this entry in the descriptor table is returned to the user. If the server returns with an error, then the global variable *errno* is set appropriately and the system call returns -1. Similarly, the system call *pipe* makes call to the pipe server for creating a pipe. If the pipe server creates a pipe successfully then it returns two ports; one for reading and the other for writing. These ports are stored in the two free slots of the descriptor table, and the indices of these entries are returned to the user.

The system calls *read/write/ioctl/close* etc., take a descriptor as one of the parameters. The port and the type of the descriptor are extracted from the descriptor table. Then a call is made to the appropriate server on this port. If the server call returns successfully, then the results are passed on to the user. Otherwise, the global variable *errno* is set appropriately and -1 is returned to the user. If some data is being sent to a server then the data is copied either in line or out-of-line depending on the size of the data. Sending the data Out-of-line is inefficient if the data size is small.

8.3 Process Related System Calls

Unix implements four major system calls for process control: *fork*, *exec*, *wait*, and *exit*. Apart from these, it provides signal related calls. *Fork* creates a new process that is a replica of the calling process. *exec* allows a process to overlay itself with a new executable file. A process may terminate by executing *exit* system call, returning exit status to its parent. The *wait* system call allows process synchronization between the parent and it's child processes. parent process can arrange to be notified by a signal when a child process exits or terminates abnormally.

8.3.1 Fork system call

In UNIX, *fork* call is the only to create new processes. The *fork* call creates a process that is identical to the the calling process. The context duplicated includes both its user-level execution state and system level state. The *fork* call returns twice, once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is zero. The parent-child relationship induces a hierarchical structure on the set of processes in the system. The new process shares all its parent's resources, such as file descriptors, signal handling status, and memory layout.

In PAWAN, a *fork* call involves the following steps:

1. A new task is created using the system call *task_create* with memory_inherit option set to TRUE. This creates a child task with all the address space shared copy-on-write. Since the descriptor table is in the task's address space it is also copied.

2. Now messages are sent to the servers requesting them to make an entry for the child and replicate the parent's server state. The process server assigns the child task a unique process identifier and sets up the parent-child relationship.

3. The send rights of all the descriptor ports in the descriptor table are given to the child task so that the child can access them with the same name (Same name since the child has got the same descriptor table as the parent task)

4. A thread is created in the child task. The state of this thread is set such that it starts executing in the fork routine itself. Then the thread is resumed.

5. Now both parent and child processes will be running in the fork routine. The parent returns the child's pid to the caller and the child process returns 0.

8.3.2 exit and wait system calls

Processes may terminate either voluntarily through an *exit* system call or involuntarily as a result of a signal. In either cases, the process termination causes the status code to be returned to the parent through the *wait* system call. The *exit* call can specify an 8 bit number to be returned to it's parent. Exit call does the following sequence of operations: All the servers are informed about the termination. The file server, pipe server and tty server close all the descriptors opened by the process and they deallocate the relevant entries. The process server stores the status and return value of exit so that the parent can get it. Rest of the process state is deallocated. Death of a child signal is sent to the parent. Similarly, the environment server deallocates the process's environment. The *wait* system call is used to synchronize its execution with the termination of a child process. The *wait* call returns with an error if there are no children running. If is finds a zombie child, then it returns the pid and the its exit status(the exit status is either the 8-bit number specified by the exit call or the signal that terminated or stopped that child process). If there is no zombie child, then the process goes to sleep.

8.4 Signal Related System Calls

In PAWAN, a process's signal related information like current mask, pending signals etc., are stored at fixed locations in its address space. A Simple locking mechanism is used to protect these data against corruption during updation. As explained in chapter 6 , in PAWAN, UNIX signalling is implemented using message passing. Every UNIX task will have one main thread called the UNIX thread and one signal thread. The signal thread will be waiting on the task's tt exception port for the signals. The *sigvec* system call is used to inform the system how a process would like a signal to be handled. The *sigvec* call specifies the signal number, the signal handler function, and the signal mask to be applied on occurrence of that signal. The *signal* call is a simplified version of *sigvec*. The implementation of *sigvec* is trivial. It sets the appropriate flags and stores the handler address in the array of handlers.

Kill system call

A Signal is posted to a process by another process using the *Kill* system call or by the kernel if the process generates an exception. The *kill* system call specifies the signal number and pid of the process to which the signal is to be sent. The em Kill routine send a message to the process server specifying the pid and the signal number. The process server checks if the sender is authorized. If so, then it sends a message to the signal port(same as the exception port) of the target task. On receiving the message, the signal thread of the target task processes the signal. If default action is to be taken then it sets a special flag and returns and the process server takes the default action. If a signal handler is to be called, then the UNIX thread is stopped, and the signal handler is called (in signal thread's context itself).

Other signal related system calls

The other signal related system calls like sigblock, sigssetmask and sigpause have also been implemented. The 4.3bsd signals provide a facility to handle signals on a separate stack. In PAWAN, the signals are always handled by a separate thread and hence on a separate stack. Hence, the call *sigstack* has not been provided.

8.5 Program Execution In PAWAN

In UNIX, every process is created by another process using the *fork* system call. *Fork* creates a child process that is a replica of the calling process. To execute a new program, *execve* system call is made which overlays the current image by a new executable program and hands over the control to it. An executable file consists of an identifying header, followed by pages of data representing the machine code (text) and initialized data. The header describes the sizes of various segments(text, data, bss), loader format, and other useful information. In PAWAN, we have provided *execve* routine that is similar to UNIX's *execve*. We have also provided a routine called *run* which creates a new process and exec's a new program in the child. This is similar to *fork* followed by *exec* in UNIX. The design and initial development was done by A.Singhai[Ashi92] and Hazel Das[Das93]. It has been ported to 396AT and has been extended to provide complete semantics. In this section, we discuss the design and implementation of the *execve*, and *run* calls.

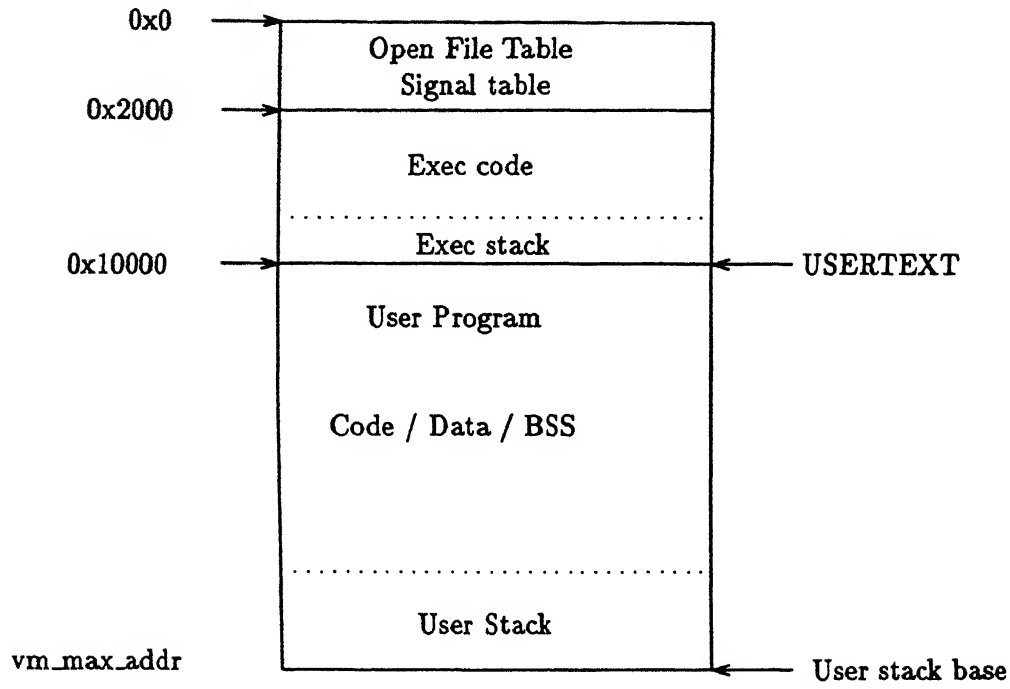


Figure 8.1: Memory Map Of A Process In PAWAN

8.5.1 Implementation of execve

UNIX implementation

The execve system call overlays the memory space of a process with a copy of the specified executable file. The contents of the user-level context that existed before the *execve* call are no longer accessible afterward except for the call parameters, which the kernel copies from the old address space to the new address space. The parameters consist of the name of the new program, the argument vector for the executable program and environment variables. There can be no return from a successful execve since the calling core image is lost. Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set. Ignored signals remain ignored, signals that are caught are reset to their default values for obvious reasons. Blocked signals remain blocked regardless of changes to the signal action. The kernel takes special action for setid programs and for process tracing. When the kernel execve's a setid program, the effective user id (or group ID) field in the process table and uarea are set to the owner ID of the file.

PAWAN Implementation of execve and run

In MACH, the parent task can read/write a child task's address space and can mark specific regions of address space for read-write sharing or copy-on-write sharing with the child [Acce86]. This feature helps us to achieve the desired functionality efficiently. The *execve* routine must be able to load the executable file into the memory. This is not possible if the code of execve itself resides in the memory. In UNIX, such a problem is not there, as the kernel does the job of loading the file onto the process's address space. Since we did not want to modify the kernel to accommodate these calls, the following strategy was employed to load the file. The file loading part of the execve code exec_code resides in a fixed region of memory with its stack lying adjacent to it. The memory map of the user process in PAWAN is shown in 8.1 . The exec_code is coded in a reentrant way and separately compiled. The exec_code will always be present in the process memory. This is inherited from the parent during a *fork/run* emulation library call. After the initial processing a non local 'goto' is made to the exec_code. The initial operations consist of checking the mode and access permissions, resetting its state in various servers, and setting up the process stack. The exec_code deallocates the current user stack and copies the new process stack into the correct location. The argument pointer and the environment pointers are pushed onto the new stack. The executable file is then loaded as specified by the header. It now jumps to the user program and starts execution.

The *run* routine creates a child process and executes the file in the child's address space. We utilize the *task_create* kernel call (with memory inheritance deactivated) provided by the MACH kernel. The parent task memory is not replicated in the child since it is anyway thrown away when we load the executable file into its memory space. The parent task requests all the trusted servers to replicate its state for the child task. Now the permission and mode of the executable file are checked. The exec_code is loaded into the fixed locations in the child's memory space. The user stack for the child is also assembled by the parent process and loaded into the corresponding location. A thread is created in the child's task. Its state is set appropriately so that it starts executing the exec_code. Now parent resumes this thread and returns. Now the child process running the exec_code loads the specified file into the memory as in the case of *execve* and hand over the control to it.

8.5.2 The Exec Server

The standalone library routines can not be used for running a file that has **setuid** or **setgid** bit set. This is because, it involves updating the effective **uid/gid**, which can be done by only a trusted server. So, a server has been provided to execute such files. In fact, the process server itself can take care of the setid exec, rather than setting up another server. The Exec Server exports the setid_exec call which is invoked by the run and execve library routines when they detects that the setuser ID or setgroup ID flag of the executable file is set. Separate calls need not be provided for execve and run since only the process which is to run the program is relevant now. The *run* routine passes on the child task created by it, while the execve routines specifies the calling task itself. At the receipt of a setid_exec request, the major task performed by the Exec Server is the updation of the associated privileges. The entries in all the trusted servers have to be updated to reflect the change in credentials. Then the server executes the program in the specified task.

8.6 Conclusion

Most of the 4.3 BSD UNIX system calls have been successfully emulated. An attempt has been made to maintain strict UNIX semantics. In many places the UNIX semantics have been redefined for various practical reasons. Now that the functionality has been achieved, we have to turn our attention to the efficiency. We have to identify the bottlenecks and take the corrective actions.

Chapter 9

Conclusion

The work described in this thesis, shows a successful implementation of Unix emulation services for MACH 3.0.

Any research in operating systems, more particularly in distributed systems, is presently influenced to a large extent by attempts to stay compatible with UNIX. Mach is no exception to this philosophy. So, the goal was to build an Operating System that has source code compatibility with UNIX. There are mainly two approaches to building operating systems on top of Mach micro kernel. Each of these approaches include the same three components: the **Mach microkernel**, one or more **system servers**, and an **emulation library**. The systems differ on how they provide the application system's personality - as a single integrated server supporting all operating system functions(the **single server** approach), or a collection of servers in which each supports an individual function of the system(the **multi-server** approach. We chose *multi-server* approach for our development . A number servers like *file servers. process server, tty server* were developed and on top of these servers, 4.3BSD UNIX emulation library was developed. This operating system, called PAWAN, now runs on two hardware platforms: a 68020 based Horizon III mini computer, and 386AT. Although PAWAN successfully emulates UNIX, it is not yet a full-fledged operating system. The standard utilities are yet to be ported. As far as the performance of PAWAN is concerned, it is quite poor compared to UNIX. Though one of the reasons for poor performance is the overheads of message passing, there is a need to look into the other bottle-necks and take corrective measures. Apart from this, the immediate need is the development of the *login server* and a *Shell.*

Bibliography

[Acce86] Accetta,M., Baron,R., Golub,D., Rashid,R., Tevanian,A., and Young,M., *MACH: A New Kernel Foundation for UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August 1986.

[Ashi92] Ashish,S., *PAWAN: A MACH based UNIX system(I)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Bach86] Bach,M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Engelwood Cliffs, May 1986.

[Bair93] Bairagi,D., *UNIX Emulation Services in PAWAN(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1993.

[Baro88] Baron,R.V., Black,D., Bolosky,W., Chew,J., Draves,R.P., Golub,D.B., Rashid,R.F., Tevanian,A.Jr., and Young,M.W., *MACH Kernel Interface Manual*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, October 1988.

[Baru92] Baruah,M., *PAWAN : A MACH based UNIX System(III)*, M. Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Bolo87] Bolosky,W., Rashid,R., Tevanian,A.Jr., Young,M., Golub,D., Baron,R., Black,D., and Chew,J., *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, July 1987.

[Cher84] Cheriton,D.R., *The V-Kernel: A Software Base for Distributed Systems*, IEEE Software, Vol 1 no.2, pp. 77-107.

[Cher88] Cheriton,D.R., *The V Distributed System*, Communications of the ACM, 31(3), March 1988.

[Coop88] Cooper,E.C., and Draves,R.P., *C Threads*, Technical.

[Coul88] Couloris,G.F., Dollimore,J., *Distributed Systems : Concepts and Design*, Addison-Wesley, July 1988.

[Das93] Das,J.J.H., *UNIX Emulation Services in PAWAN(I,)* M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1993

[Dean90] Dean,R., Golub,D., Forin,A., Rashid,R., *UNIX as an Application Program*, Proceedings of the 1990 Summer Usenix, USENIX, June 1990.

[Drav89] Draves,R.P., Jones,M.B., and Thompson,M.R., *MIG: The MACH Interface Generator*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1989.

[Gold89] Goldstien,I., *Introduction to th MACH Development workshop*, Proceedings of the Workshop on MACH Development, Boston, 1989.

[Golu87] Golub,D.B., Tevanian,A.Jr., Rashid,R., Black,D.L., Cooper,E., and Young,M.W., *MACH Threads and the UNIX Kernel: The Battle for Control*, Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

[Gopa92] Gopal,B., *PAWAN: A MACH based UNIX System(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[guil91] Guillemont,Marc., *CHORUS: A Support For Distributed And Configurable Ada Software*, chorus systemes, 1991.

[Jone86] Jones,M.B., Rashid,R.T., *MACH and Matchmaker : Kernel and Language Support for Object Oriented Distributed systems*, ACM Sigplan Notices, Vol. 21, no.11, pp. 67-77.

[Juli92] Julin,D.P., Chew,J.J., Stevenson,J.M., Guedes,P., Neves,P., Roy,P., *Generalised Emulation Services for MACH3.0: Overview, Experiences and Current status*, MACH Symposium, USENIX Association.

[Leff89] Leffler,S.J., McKusick,M.J., Karels,M.J., Quarterman,J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., May 1989.

[Mull85] Mullender,S.J., Tanenbaum,A.S., *A distributed File Server Based on Optimistic Concurrency Control*, Proceedings of the Tenth ACM Symposium on Operating Systems, Dec. 1985.

[Pete90] Peterson,L.M., Hutchinson,N.C., O'Malley,S.M., and Rao,H.,C., *The X-kernel: A Platform for Accessing Internet Resources*, IEEE Computer, 23(5), May 1990.

[Rao92] Rao,P.V., *PAWAN : A MACH based UNIX System(IV)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Rash86] Rashid,R.F., *Threads of a New System*, UNIX Review August 1986.

[Rene89] Renesser,R.V., Tanenbaum,A.S., *The Evolution of a Distributed Operating System*, LNCS, Vol 433, Springer-Verlag.

[Tane90] Tanenbaum et al., *Amoeba - A Distributed Operating System for the 1990's*, IEEE Computer, Vol. 18 No. 2, 1990.

[Teva87] Tevanian,A.Jr., and Rashid,R., *MACH: A Basis for Future UNIX Development*, Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University.

[Walk83] Walker et al., *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, October 1983.